# CS 166 Homework 1

Leo Martel (lmartel)

4/9/2014

**Question 1** (Prrgmrr)**.** *Not yet submitted*

**Question 2** (Flexible Sequences)**.**

**Overview:** We use an Order Statistic tree with a modified insertion scheme (rather than sorting by value, we explicitly choose the position of each node as we insert them).

**Representation:** A Red/Black tree (or any balanced BST with guaranteed $O(\lg n)$ height) that tracks the following information (in addition to left/right child) at every node n:

- $n.l$: number of items in left subtree
- $n.r$: number of items in right subtree
- $n.t$: $1 + n.l + n.r$

We proved in lecture that maintaining this augmentation does not change the runtime of any BST operations, and enables $select(k)$ and $rank(v)$ in $O(\lg n)$ time.

**Operations:**

- seq.insert(i, x): run $select(i)$ to find the $i$th order statistic (call it node $n_i$), and insert a new $n_x = node(x)$ between $n_i$ and its in-order predecessor as normal. Select runs in $O(n)$ time and insert runs in $O(\lg n)$ time for a total runtime of $O(\lg n)$.
- seq.delete(i): run $select(i)$ to find the $i$th element, then delete it as normal. $O(\lg n)+O(\lg n) = O(\lg n)$ total runtime.
- seq.lookup(i): run $select(i)$ and return the value found. $O(\lg n)$ time.
- seq.set(i, x): run $select(i)$ and replace the value in the found node with $x$. Since the tree is not sorted by x-values no rebalancing is necessary, which means the $O(\lg n)$ select operation is the total runtime.
- seq.size(): $root.t$ is the number of elements in the sequence. $O(1)$.
- split(seq, i): run $select(i)$ and run the normal BST split operation at that node, giving us $seq_1$ and $seq_2$ back. We can quickly fix the additional fields by recalculating the values for all (ex-)ancestors of the (ex-) $i$th node. Since the fields can be calculated locally we know only ancestors of $n_i$ need to be recalculated; since it has $O(\lg n)$ ancestors this recalculation takes $O(\lg n)$ time. The split took $O(\lg n)$ time, giving us a total runtime of $O(\lg n)$
- concat($seq_1, seq_2$): run $seq_1.delete(seq_1.size() - 1)$ to retrieve the last element k of $seq_1$ in $O(\lg n)$ time, then run $join(seq_1, k, seq_2)$ as normal. Similarly to above, only new ancestors of $k$ will need to be updated, so the updates will take $O(\lg n)$ time. The join takes $O(1+|h_1-h_2|) \leq O(1+\lg n) = O(\lg n)$ for significant n, so our total runtime is $3O(\lg n) = O(\lg n)$.

And that's everything.

**Question 3** (Dynamic Maximum Overlap)**.**

i. **Overview:** we make a sorted list of event start times and a sorted list of event end times, and iterate through to find the maximum (num started - num ended) at any point in time.

**Representation:** we use two arrays, $[s_i]$ and $[t_i]$, a cursor in each array, an int maxOverlap, and an int $\Delta$.

**Algorithm:** first, copy the start times into one array and the end times into another, then sort both in ascending order. Runtime so far: $O(n) + O(n) + O(n \lg n) + O(n \lg n) = O(n \lg n)$ assuming an optimal comparison sort. Start both cursors, $s_c$ and $t_c$ at position $-1$.

Iterate the following until all start times have been visited:

Move $s_c$ forward one spot, and increment $\Delta$, representing a new event starting. Repeat (Move $t_c$ forward; decrement $\delta$) while the next end time $t_j$ is less than or equal to (since we're using open intervals) the current start time $s_i$, representing events that ended before $s_i$ began.

Now, record $\Delta$ in *maxOverlap* if it's a new maximum.

Each iteration takes only constant-time operations, and we always increment our start-time cursor once per iteration, so we do $O(n)$ work in this part of the algorithm. The total runtime is $O(n \lg n) + O(n) = O(n \lg n)$.

**Proof of correctness:** at any start time $s_i$, we have "started" all events that start before $s_i$, because we visit the start times in sorted order. Similarly, we have "ended" all events that end before $s_i$, because we visit end times until we can't anymore without surpassing $s_i$. Observe that the number of simultaneous events at any point in time is the number of events that have started but not ended. In addition, this number only increases at the moment an event starts; thus, the maximum will occur at one of these moments. We do examine all the start times, ensuring we get the correct maximum.

ii. **Overview:** we use a augmented balanced BST of times (storing start and end times separately). We can separate start and end times without changing the max overlap, as proved in (i).

**Representation:** a standard balanced BST (like a Red/Black tree) with the following information at each node:

- n.value: +1 for start times and -1 for end times
- n.total: the sum of values of all the nodes in the subtree rooted at n
- n.max: the maximum overlap of the subtree rooted at n, starting at the first time in the subtree

**Calculating n.total:** $n.total = n.left.total + n.value + n.right.total$, calculated in $O(1)$ time by examining child nodes (so it can be maintained without changing BST runtimes). Proof of correctness is trivial.

**Calculating n.max:** $n.max = max(n.left.max, n.left.total+n.value, n.left.total+n.value+n.right.max)$, calculated in $O(1)$ time by examining child nodes (so it can be maintained without changing BST runtimes).

**Note:** mean that n.total and n.max can be maintained through insertions and deletions

**Proof of correctness of n.max:** by induction over h (height of tree).

Base case: h = 2 (one event), with root node r and child c.

Case 1: $r.value = +1, c.value = -1$. c must be the right child of r, since start times must follow end times.

$$c.max = max(0, 0-1, 0-1+0) = 0; r.max = max(0, 0+1, 0+1+0) = 1$$

Case 2: $r.value = -1, c.value = +1$. c must be the left child of r.

$$c.max = (0, 0+1, 0+1+0) = 1; r.max = max(1, 1-1, 1-1+0) = 1$$

These cases are exhaustive because the total value of the tree must be 0 (all that starts must come to an end) and the start time must precede the end time. In both cases, $r.max = 1$, which is the correct max overlap of a single event.

Inductive step: assume that $root.max$ is correct for trees of height k. Let $r$ be the root of a tree of height $k+1$.

First, observe that $r.left.max$ and $r.right.max$ are correct, since their respective subtrees are of height $k$.

Next, recall from part $(i)$ that the max overlap is the maximum sum of values ($\pm 1$) from the first time $n_0$ to some time $n_m$.

Case 1: $n_m$ is contained in the left subtree. Thus the max overlap of the left subtree is the max overlap of the whole tree, so $r.left.max$ gives us the correct max overlap.

Case 2: $n_m = r$. Thus the max overlap is the sum of values from $n_0$ to $r$. Since all times before $r$ are contained in its left subtree, $r.left.total + r.value$ gives us the correct max overlap.

Case 3: $n_m$ is contained in the right subtree. The max overlap is the $sumOfValues[n_0, n_m]$, which can be broken down as follows:

$$sumOfValues[n_0, r] + sumOfValues[succ(r), n_m]$$

It's obvious that

$$sumOfValues[n_0, r] = r.left.total + r.value$$

Next, by lemma 1, we know that the optimal $n_m$ within the whole tree is the same as the optimal $n_m$ within the right subtree. We also know that the leftmost value in the right subtree is $succ(r)$ by BST definition. Thus,

$$r.right.max = sumOfValues[succ(r), n_m]$$

Adding these two results together gives us

$$max\ overlap = sumOfValues[n_0, n_m]$$

$$= sumOfValues[n_0, r] + sumOfValues[succ(r), n_m]$$

$$= r.left.total + r.value + r.right.max$$

These 3 cases are exhaustive ($n_m$ has to be somewhere), so the max overlap will be one of these 3 results,

$$n.left.max, n.left.total + n.value, n.left.total + n.value + n.right.max$$

Since $r.max$ is exactly the max of these three options, $r.max$ is correct. QED.

Lemma 1: if the optimal $n_m$ for the whole tree is contained within the right subtree, then it's equal to the optimal $n_m$ looking at only the right subtree (so, starting at $succ(r)$ instead of $n_0$).

Proof of Lemma 1: by contradiction. Assume $n_m$ is optimal for the big tree but $n_q$ is optimal for the little tree. Thus the max overlap for the entire tree is $sumOfValues[n_0, n_m]$ but $sumOfValues[succ(r), n_m] < sumOfValues[succ(r), n_q]$. This means $sumOfValues[0, n_q] > sumOfValues[n_0, n_m]$, contradicting the optimality of $n_m$. QED.

**Now that we've proved all that...**

- insert(s, t): insert s and t separately into the augmented BST. $2O(\lg n) = O(\lg n)$.
- delete(s, t): delete s and t separately (if duplicate start/end times exist, ensure that the s node has value $+1$ and the t node has value $-1$). $2O(\lg n) = O(\lg n)$
- max-overlap(): return $root.max$. $O(1)$

As shown above, $root.max$ is correct and can be maintained through log-time deletions and insertions, so this all works. We're done!